A Replica Management Service for High-Performance Data Grids

The Globus Data Management Group

http://www.globus.org/datagrid

1 Introduction

In many scientific disciplines, a large community of users requires remote access to large datasets. An effective technique for improving access speeds and reducing network loads can be to replicate frequently accessed datasets at locations chosen to be "near" the eventual users. However, organizing such replication so that it is both reliable and efficient can be a challenging problem, for a variety of reasons. The datasets to be moved can be large, so issues of network performance and fault tolerance become important. The individual locations at which replicas may be placed can have different performance characteristics, in which case users (or higher-level tools) may want to be able to discover these characteristics and use this information to guide replica selection. And different locations may have different access control policies that need to be respected.

These considerations motivate this proposal for a replica management system charged with managing the copying and placement of files in a distributed computing system so as to optimize the performance of the data analysis process. Our goal in designing this service is not to provide a complete solution to this problem but rather to provide a set of basic mechanisms that will make it easy for users, or higher-level tools, to manage the replication process.

Our proposed replica management service provides the following basic functions:

- The registration of files with the replica management service.
- The creation and deletion of replicas for previously registered files.
- Enquiries concerning the location and performance characteristics of replicas.
- The updating of replicas to preserve consistency when a replica is modified.
- Management of access control at both a global and local level.

The proposed service builds on components provided by the Globus Toolkit, specifically the public-key-infrastructure-based Grid Security Infrastructure, the Replica Catalog Service, the Grid Information Service, and the GSI-FTP extensions to FTP.

In this document, we first describe the requirements of two different application domains, high-energy physics and climate modeling, which motivate the replica management service design. We then provide first a detailed description of a set of low-level Phase 1 replica management functions, followed by some early thoughts on more sophisticated higher-level Phase 2 services.

A note on the word "replica": The word *replica* has been used in a variety of contents with a variety of meanings. For example, it is sometimes used to mean "a copy of a file that is guaranteed to be consistent with the original, despite updates to the latter." For the purposes of this document, we define a replica to be simply a *managed copy of a file*. The replica management system controls where and when copies are created, and provides information about where copies are located. However, the system does *not* make any statements about file consistency. In other words, it is possible for copies to get out of date with respect to one another, if a user chooses to modify a copy.

2 Motivating Examples

We present two examples of application domains in which we believe our replication service can be useful.

2.1 Objectivity Databases in Physics Experiments

Particle physics experiments are characterized by the need to perform analysis over large amounts of data. To enable the selection of the data of interest, and to simplify the development of analysis codes, several such experiments (ATLAS, BaBar, CMS), have selected object-oriented technology as a structured file representation for storing the physics data to be analyzed. Users at many sites worldwide then need to be able to access data contained in these databases.

2.1.1 Use of Objectivity

In the physics experiments of interest, Objectivity (http://www.Objectivity.com/) is the database technology that has been selected for data storage. Objectivity stores collections of object in a single file called a *database.* Databases can be grouped into larger collections called *federations.* Objects in one database can refer (point) to objects in another database in the same or a different federation. In the physics experiments we are considering, each database file is several gigabytes in size. Federations are currently

limited to 64K files, however, future versions of Objectivity will eliminate this restriction. Some experiments plan to exploit this feature and anticipate creating federations with millions of individual database files.

There are two types of data generated by physics experiments:

- Experimental data that represents the information collected by the experiment. There is a single creator of this data, and once created, it is not modified. However, data may be collected incrementally over a period of weeks.
- Metadata that captures information about the experiment (such as the number of events) and the results of analysis. Multiple individuals may create metadata. The volume of metadata is typically smaller than that of experimental data.

The consumers of the various types of data can number in the hundreds or thousands. Because of the geographic distribution of the participants in a particle physics experiment, it is desirable to make copies of the data being analyzed so as to minimize the access time to the data. This replication is complicated by several factors, e.g.:

- Complete data sets can be very large. Thus one may need to replicate only "interesting" subsets of the data. However, because of the way Objectivity is being used by the various physics experiments, the subsets of the data that need to be updated may span many database files, or even many federations.
- Database files may be modified. One cause for this is that the write time into database files can be quite large. In some experiments (Babar, http://www.slac.stanford.edu/BFROOT/), it can take several weeks to collect an entire data set. However, one would like to make data available incrementally, potentially every few days.

One approach to this distribution problem would be to use existing Objectivity methods for distributing database files in a federation across multiple machines. However, Objectivity was not designed to operate in the regime of wide-area, very high performance networks. Discussions with engineers within Objectivity (Harvey Newman, personal communication) confirm that this is not a recommended approach. Hence, we believe that our replica management service may represent a better approach.

2.1.2 Inter-Database References

One issue that has the potential to complicate the replication process is that Objectivity keeps track of where objects exist within a federation by building a catalog. It is possible to move a database from one federation to another (e.g., federations at different sites) by performing an *export* from the first federation followed by an *import* into the second. We note that maintenance of the catalog depends on the name of the database file, and not the contents. Therefore catalog operations can take place concurrently with data movement operations.

One potential problem in the export/import strategy is the existence of cross database pointers. Five distinct strategies have been proposed to address this problem:

1. Ignore the problem, with the knowledge that cross database links may be broken in a replicated file.

2. Identify cross-database references, and null them out, creating a self-contained database file, with the loss of some information.
3. Insure that whenever any database file is moved, all of the databases that may be referred to by this database are moved as well. This requires using database schema to identify the complete set of inter-dependent files.
4. Generate an intermediate database file that contains all objects of interest, including external references. This method can be viewed as an optimization of the previous method, with the advantage of reducing the amount of data that may be transferred. The disadvantage of this method is increased complexity in keeping track of replica location and potentially reducing the effectiveness of data caches.
5. Replace the interobject references with "remote references" that can be resolved via a (slow) inter-site object access mechanism. (This is basically a variant of #1, in which we have a global name space for object identifiers and can use the catalog to work out where "nonlocal" databases are located. This is what BaBar does, for example.)

We note that all five methods are equivalent from our perspective: if any of them is adopted, then all can employ our proposed replication mechanism, which enables the copying of a set of databases files from one location to another.

2.1.3 Database Updates

Another potentially complicating issue is that while in principle the primary databases produced by physics experiments are read-only (once created, their contents do not change), we find in practice that during initial production of the data (a period of several weeks) database files change as new objects are added. Users want access to these database files during this production period, so we face a need for updating of replicated database files. In addition, metadata may be either modified, or augmented over time.

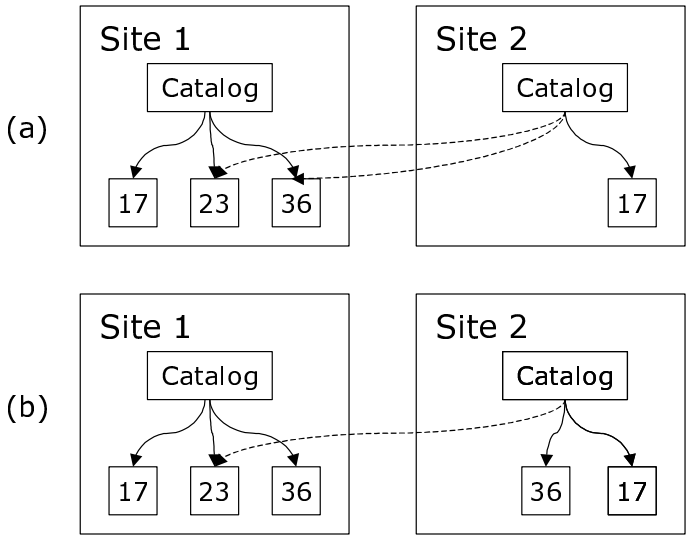
Different experiments tend to use Objectivity in different ways and hence have somewhat different requirements in this area. We explain the requirements in two experiments, BaBar and CMS.

2.1.3.1 BaBar

In BaBar, objects are appended to the databases in a federation over the course of several weeks. The practical impact of these *logical* append operations is that many database files can be changing simultaneously during this period, as they fill up, after which they do not change further. Individual database files are around ten GB in size, currently, due to an Objectivity limit of 64K files, due to be removed at the end of 2000; the goal is to reduce this size to O(1) GB, in which case there will be millions of files. In the latter case, updates to files during the production phase will become less of a problem. Metadata files are currently 2 GB in size and will also become smaller.

The following figure shows the sort of database structure used in BaBar. Different sites maintain identical catalogs, with some catalog entries referencing local copies of database files and others referencing remote copies. In (b), we see the result of replicating

database file 36 at Site 2: the file is copied and the catalog at Site 2 is updated to reference the local copy.



A master-slave model is used to propagate modifications. During the production period, updated (whole) files are transmitted to destination locations once a week. They would like these updates to occur more frequently: say once every three days. The frequency of updates is constrained by trans-Atlantic bandwidth.

A master-slave model is also used to control update access to databases containing metadata. These semantics are provided by partitioning the object identifier space, so that each participating site has exclusive write access to a predefined subset of objects in the federation.

2.1.3.2 CMS

The CMS collaboration has adopted a different approach to the use of Objectivity, in which data files do not change once created. However, metadata files *do* change to reflect the increasing total number of events in the database. Metadata updates need to be propagated to all replicas.

The CMS collaboration are interested in supporting distribution of "partial databases" containing only those objects of interest to a particular scientist. We do not address this requirement in this document.

2.2 Climate Model Data

In the climate modeling community, modeling groups sometimes generate large "reference simulations" that are then of interest to a large international community. The output from these simulations can be large (many Terabytes). The simulation data is typically generated at one or more supercomputer centers and is then "released" in stages to progressively larger communities: first the research collaboration that generated the data, then perhaps to selected colleagues, and eventually to the entire community.

In contrast to the physics community, data is not maintained in databases but rather as flat files, typically structured using for example NetCDF, with associated metadata. In addition, files are not updated once released. However, we believe that the basic requirements for data distribution (replication) are sufficiently similar that a common replica management service can be employed.

3 Building Blocks

Our proposed replica management service builds on two elements of the Globus data management architecture:

1. Replica catalog, and associated APIs, for keeping track of the location of replicas. This catalog maintains a mapping from a logical (i.e., global) file name to one or more physical file names on different storage systems. For the API, see http://www.globus.org/datagrid/deliverables/globus_replica_catalog/.
2. GridFTP, an FTP-based transport protocol (and associated APIs) for moving files between two endpoints. The protocol and APIs are extensible. For example, we can supply plug-ins that implement application-specific error handling procedures, such as retries. Support for parallelism and striping is also planned. See http://www.globus.org/datagrid/deliverables/ for details

In addition, we can take advantage of the following additional Globus services when developing replica management functions:

3. The Grid Security Infrastructure (GSI) protocol and tools, used to provide single-sign-on, public-key authenticated access to remote data and computers. GSI-FTP uses this.
4. The Globus Resource Allocation and Management (GRAM) protocol for accessing remote computation. We can use this, for example, to invoke remote cataloging operations following successful completion of a replica creation.
5. The Grid Information Service (GIS), which provides Lightweight Directory Access Protocol (LDAP) to information about the structure and state of storage systems, computers, and networks

4 Proposed Replica Management Solution

We propose to develop a replica management service that will meet the requirements listed above, i.e., Objectivity files in particle physics experiments as well as climate model simulation output. Due to the complexity of this task we propose a staged approach. In Phase 1, we will focus on providing “dumb” but efficient and reliable data replication functions:

- Replica management functions: We will build on the Globus replica catalog and GridFTP data transport services described above to provide basic functions for creating, updating, locating, and deleting replicas of entire files. These basic functions are “dumb”: they simply allow the user to request that a file be copied

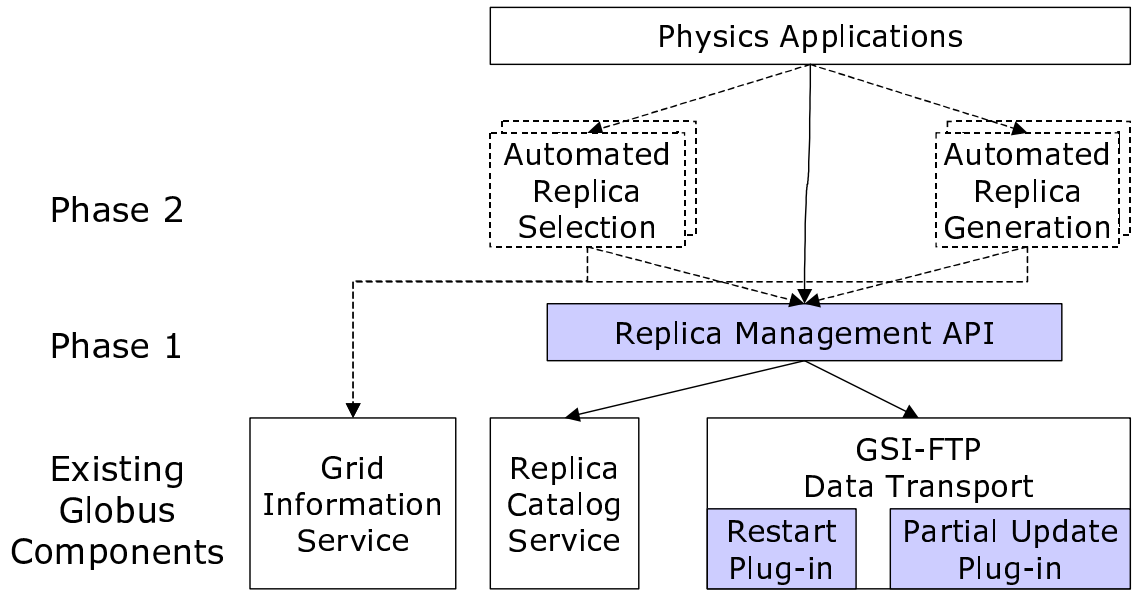
from location A to location B. That is, they do not support any logic for determining whether, when, where, or from where a replica should be created.

- Data transport extensions for failure recovery and updating transfers: Exploiting the extensibility of the GridFTP infrastructure, we will build an error handling module that supports retries and restarts in the event of link failures, plus an update-specific transport that uses checksums to reduce the amount of data transferred when updating a file.

These functions will make it possible to develop a variety of innovative functions during a subsequent Phase II, for example automated replica selection functions. The definition of these functions may well vary from application to application.

We now proceed to describe first the replica management functions and then the data transport extensions that we propose to develop during Phase I. Some initial thoughts on Phase II work are described in Section 5.

The following figure shows the general structure of the proposed system.



4.1 Assumptions

In designing the Globus replica management API, we made the following decisions and assumptions:

- This is an API for functions that provide the core functionality necessary to do replica management. It does not contain all of the bells and whistles that an end user might want in a replica management API. Those features can be layered on top of this API later.
- This API provides functions for managing replication for individual files. It does not contain functions for managing multiple files. However, the latter can be efficiently built on the former, as care was taken in this API to allow for caching of

session state across several single-file operations. The advantages of the single file approach is that it simplifies the API and implementation, and it allows for various multi-file approaches to be built on top of this basic API.

- The replica catalog must remain consistent at all times, including during copy operations and after a failure. In other words, users of the catalog must continue to be able to use the catalog, even during a replica management operation, or after a replica management operation fails.
- If a replica management operation fails, all information necessary to roll back the operation should be stored in the replica catalog.
- In order to provide consistency and rollback, we have introduced a "rollback lock" into the globus_replica_catalog library. This object class must be of type "GlobusReplicaLocationFileLock", or an extension of that object class. It lives in the DIT under the location entries. At the beginning of any replica management operation that modified a location file, the operation does an ldap_add of a file lock entry under the appropriate location. Since ldap_add is atomic, if the add fails then we have acquired the lock, and if it fails then we know somebody else is currently working on the file. This file lock object class has a required attribute of "timeout", which defines how long the lock is good for in GMT – this will allow the lock to be safely broken after this timeout, particularly in the event of a failure. If a replica management operation takes longer than the timeout, then it must periodically ldap_modify the entry with a new timeout. In addition, this object class will be extended by the globus_replica_management library to contain rollback information. Before the replica management library does anything that might need to be rolled back in the case of failure, it ldap_modifies the entry with enough information to allow anyone to perform the rollback in case of failure.
- The management functions provide for concurrency control (i.e. through advisory locking) amongst multiple users of the replica management API.
- A "Community Authorization Server" (CAS) is being defined in a separate document. A CAS can optionally be used by this replica management library to provide for easier management authentication and authorization amongst a multi-institutional community of users.

The subsequent subsections define each function of the API in more detail. Briefly, there are functions for:

Session management: There are a set of functions for creating, configuring, and destroying session handles. This allows, for example, caching of connection states to GridFTP and LDAP servers, so that multiple files can be managed efficiently through this API.

Catalog creation: There are a set of functions for creating and populating a replica catalog collection, and storage system locations within the collection.

File maintenance: There are functions for copying, updating, deleting, and checking status of files amongst locations within a collection.

We will also provide functions for controlling who can access, and make replicas of, individual files and collections of files.

In addition, we note that the underlying replica catalog API provides us with the following useful function:

globus_replica_catalog_location_search_filenames: Return the locations of a specified logical file (or files).

Notice that none of these functions are not specific to any specific application: they simply manage the movement and registration of copies of files. In the case of Objectivity-based applications, it is the responsibility of the application to identify the objects that need to be managed, map the object identifiers to one or more database files, and then invoke our functions to access or copy those files.

4.2 Session Management

The following subsections describe functions to create, configure, and destroy session handles. A handle has the type **globus_replica_management_handle_t**, and must be threaded through the other replica management functions. It contains configuration and state information about the management operation in progress, and allows for session state (e.g. connections to GridFTP and LDAP servers) to be cached between management operations.

A handle attribute structure, **globus_replica_management_handleattr_t**, can be used to configure a handle, for example, with security information, performance tuning hints, etc. The set of handle attribute options will grow over time, including attributes that are inherited from the globus_ftp_client and globus_replica_catalog APIs.

4.2.1 globus_replica_management_handleattr_init()

globus_result_t
globus_replica_management_handleattr_init(
*globus_replica_management_handleattr_t * handleattr);*

Initialize a handle attribute structure.

4.2.2 globus_replica_management_handleattr_destroy()

globus_result_t
globus_replica_management_handleattr_destroy(
*globus_replica_management_handleattr_t * handleattr);*

Destroy a handle attribute structure.

4.2.3 globus_replica_management_handleattr_set_cas()

globus_result_t
globus_replica_management_handleattr_set_cas(
*globus_replica_management_handleattr_t * handleattr,*
*char * cas_url);*

Set the Community Authorization Server (CAS) field in the handle attribute.

4.2.4 globus_replica_management_handleattr_get_cas()

globus_result_t
globus_replica_management_handleattr_get_cas(
*globus_replica_management_handleattr_t * handleattr,*
*char ** cas_url);*

Get pointer to the Community Authorization Server (CAS) information from the handle attribute. The returned cas_url must not be modified by the user, and should not be freed.

4.3 Catalog Creation

The following subsections describe a set of functions for creating replica catalog collections, creating locations within a collection, and populating that collection and location with new or existing files.

4.3.1 globus_replica_management_filespec_init()

globus_result_t
globus_replica_management_filespec_init(
*globus_replica_management_filespec_t * filespec);*

Initialize a file specification structure.

A file specification is a set of one or more file expressions, which name a set of files. In the initial implementation, a file specification is used to help maintain access control policies, while in later implementation it will likely be extended to use by the replica catalog to more concisely express large collections.

4.3.2 globus_replica_management_filespec_add()

globus_result_t
globus_replica_management_filespec_add(
*globus_replica_management_filespec_t * filespec,*
*char * file_expression);*

Add the expression to the file specification. This expression defines file names within a context, where the context is defined in whatever function uses the filespec. For example, a context might be a particular directory within a particular storage server, so the filespec defines a set of files within that directory.

The expression is simply a filename, with optional wildcards of "*". The following are example expressions:

- "*":all files within the context.
- "file*": all files whose names start with "file" within the context
- "dir/*": all files in the "dir" subdirectory within the context
- "dir*/*": all files within any directory whose name starts with "dir"

- This expression language will be improved in the future.

4.3.3 globus_replica_management_filespec_destroy()

globus_result_t
globus_replica_management_filespec_init(
*globus_replica_management_filespec_t * filespec);*

Destroy the file specification.

4.3.4 globus_replica_management_collection_create()

globus_result_t
globus_replica_management_collection_create(
*globus_replica_management_handle_t * handle,*
*char * collection_url,*
*char * objectclass);*

Create an empty collection (i.e. with no filenames in it), with default access control rights.

The caller must have the authorization to create the appropriate collection object in LDAP.

4.3.5 globus_replica_management_location_create()

globus_result_t
globus_replica_management_location_create(
*globus_replica_management_handle_t * handle,*
*char * collection_url,*
*char * location_name,*
*char * objectclass,*
*char * root_dir_url,*
*globus_replica_management_filespec_t * filespec);*

Create an empty storage system location (i.e. with no files in it) within a collection.

The caller must have authorization to create the storage system directory, root_dir_url, and to add a location to the replica catalog collection.

Parameters:

- handle: The session handle.
- collection_url: The LDAP URL of the collection in which to add the new location
- location_name: The name to use for this location within the replica catalog. This name must be unique within this collection.
- objectclass: The LDAP object class to use for the location object in the replica catalog. This must either be “GlobusReplicaLocation”, or the name of an object class that extends “GlobusReplicaLocation”.

- root_dir_url: This is an URL which is the directory in the storage system in which the files of this collection will reside. This becomes the "URL constructor" of the location. For example, "gsiftp://host.org/directory".
- filespec: A specification of what files can be named within the root_dir_url.

Approach:

- If a CAS is defined for this handle, then connect to the CAS, and create a new access control entry for <root_dir_url>/<filespec>, with default access rights. The CAS will verify that this new entry does not violate global access control policies of the community, and does not clash with any existing access control policies. Then get an authorization credential from the CAS to modify the replica catalog collection, and to create and access the storage system root_dir_url.
- Create or verify the existence of the storage system directory defined by root_dir_url.
- Add the location object to the replica catalog collection.

4.3.6 globus_replica_management_file_register()

globus_result_t
globus_replica_management_file_register(
*globus_replica_management_handle_t * handle,*
*char * collection_url,*
*char * location_name,*
*char * filename);*

Register an existing file from storage server into a collection and location.

This function is used to build a replica catalog collection. It does *not* copy the file. Rather, the file already exists on a storage system which is referenced by a replica catalog location. The "filename" parameter is interpreted relative to the url constructor from the replica catalog location. In other words, the complete url for the file is derived by combining the url constructor from the replica catalog location specified by location_name parameter, with the filename parameter.

The filename is added to both the replica catalog location and collection objects, as required.

The caller must have authorization to add a filename to the replica catalog collection and location, to read the file from the storage system.

Parameters:

- handle: The session handle.
- collection_url: The LDAP URL of the collection in which to add the new file. This collection must already exist before this function is used, for example by calling globus_replica_management_collection_create().

- location_name: The name of the location within the replica catalog to which the file should be added. This location must already exist before this function is used, for example by calling globus_replica_management_location_create().
- filename: The filename of file to add. This file must exist within the storage system directory referred to by the url constructor of this replica catalog location. The filename must also match the file specification for this replica catalog location.

Approach:

- If a CAS is defined for this handle, get an authorization credential from the CAS to query and modify the replica catalog collection and location.
- Query the replica catalog to get the url_constructor for the replica catalog location.
- If a CAS is defined for this handle, get an authorization credential from the CAS to read the file "url_constructor/filename".
- Verify that the caller can read the url_constructor/filename from the storage system. If this fails, then return error with no rollback necessary.
- If the filename is not already part of the collection, then add it to the collection. If this fails, then return error with no rollback necessary.
- If the filename is not already part of the location, then add it to the location. If this fails, then return error with no rollback necessary. (Question: Do we need to rollback the collection addition in this case? It won't harm anything to leave it there. And if we do want to rollback, then we need to introduce new locking structures at the collection level, and not just at the location level.)

4.3.7 globus_replica_management_file_publish()

globus_result_t
globus_replica_management_file_publish(
*globus_replica_management_handle_t * handle,*
*char * collection_url,*
*char * source_url,*
*char * dest_location_name,*
*char * filename,*
globus_replica_management_overwrite_behavior_t overwrite_behavior);

Publish a file into a collection and location, by copying that file from its source into the storage system and updating the replica catalog.

This function is used to build a replica catalog collection. But unlike globus_replica_management_file_register(), this function *does* copy the file from its existing location into the storage system. The source file is passed as a parameter, and the destination is derived by combining the url constructor from the replica catalog location specified by dest_location_name parameter, with the filename parameter.

The filename is added to both the replica catalog location and collection objects, as required.

This function can be called at a location distinct from the source and destination file systems.

Throughout the operation of this function, and even in the event of failure, the replica catalog will remain consistent. That is, any files that are listed in a replica catalog location are complete and uncorrupt (barring spontaneous corruption of a file by a storage system). Pending operations on files are locked, and their state is checkpointed to the replica catalog, so that in the event of failure this function will leave rollback/cleanup information in the replica catalog. This information can then be used by either the same or a different client to rollback/cleanup any debris left around by the failure.

The caller must have authorization to add a filename to the replica catalog collection and location, and to write the file to the storage system. The caller must also be able to read the source file using his/her own credentials.

A return value of GLOBUS_SUCCESS indicates that the function succeeded. Any other value indicates failure, in which case the globus_error functions can be used to inquire for more information about the cause of the error.

Parameters:

- handle: The session handle.
- collection_url: The LDAP URL of the collection in which to add the new file. This collection must already exist before this function is used, for example by calling globus_replica_management_collection_create().
- source_url: The file that is to be published into the replica catalog, and copied into the replica catalog location's storage system. The url can have one of the following schemes: "file:", "ftp:", or "gsiftp:".
- dest_location_name: The name of the location within the replica catalog to which the file should be added. This location must already exist before this function is used, for example by calling globus_replica_management_location_create().
- filename: The filename that this file should be given within the location's storage system. The filename must also match the file specification for this replica catalog location.
- overwrite_behavior: This parameter specified the behavior to take if the file already exists in the storage system defined by the location. The options are:

  GLOBUS_REPLICA_MANAGEMENT_OVERWRITE_DISALLOW: If the file already exists, then this function should return an errors, and not modify either the file or the replica catalog.

  G LOBUS_REPLICA_MANAGEMENT_OVERWRITE_SAFE: If the file already exists, then copy the file to the storage system using a temporary name, and then rename it when the copy is complete. If a failure occurs, then the original file will be left intact. The original file will also remain available via the replica catalog during the transfer.

G LOBUS_REPLICA_MANAGEMENT_OVERWRITE_UNSAFE: If the file already exists, then copy the file to the storage system over the top of the existing file. If a failure occurs, then the original file will be corrupted, and the rollback will attempt to remove the original file. The file will be removed from the replica catalog prior to starting the copy, and reinstated in the catalog after the copy completes.

Approach:

- Verify that the caller can read source_url, using his/her own credential.
- If a CAS is defined for this handle, get an authorization credential from the CAS to query and modify the replica catalog collection and location.
- Query the replica catalog to get the url_constructor for the replica catalog location.
- If a CAS is defined for this handle, get an authorization credential from the CAS to read and write the file "url_constructor/filename".
- Verify that the caller can cd to the url_constructor directory on the storage system. If this fails, then return error with no rollback necessary.
- Create lock (no rollback info) for filename under location entry. If this fails, return error with rollback. (Rollback may be needed, if lock entry was created in LDAP database, but the net dropped during the response.)
- Check to see if "url_constructor/filename" exists.
  If it does not exist, then:
  - set destfile to filename
  - modify the rollback lock (remove destfile or restart transfer). If this fails, return error with rollback.
  If it does, and if overwrite_behavior is DISALLOW, then return error with no rollback necessary.
  If overwrite_behavior is SAFE, then:
  - set destfile to filename.update
  - modify the rollback lock (remove destfile or restart transfer). If this fails, return error with rollback.
  If overwrite_behavior is UNSAFE, then:
  - set destfile to filename
  - modify the rollback lock (remove destfile or restart transfer). If this fails, return error with rollback.
  - modify the location entry to remove the filename. If this fails, return error with rollback.
- Perform third party transfer from source_url to destfile. If this fails, return error with rollback.
- During the transfer, periodically:
  - modify the rollback lock with restart progress and transfer performance information
  - callback user function with progress and performance information

- If the filename is not already part of the collection, then add it to the collection. If this fails, then return error with rollback.
- If filename existed, and overwrite_behavior is SAFE, then rename destfile to filename. If this fails, return error with rollback.
  Else add the filename to the location. If this fails, then return error with rollback.
- Remove the lock. If this fails, then return error with rollback.

4.4 File Maintenance

4.4.1 globus_replica_management_file_copy()

globus_result_t
globus_replica_management_file_copy(
*globus_replica_management_handle_t * handle,*
*char * collection_url,*
*char * source_location_name,*
*char * dest_location_name,*
*char * filename,*
globus_replica_management_overwrite_behavior_t overwrite_behavior);

Copy a file from one replica catalog location to another, and update the replica catalog destination location.

The parameters and approach of this function is the same as for globus_replica_management_file_publish(), except that the source of the file in this case is determined from the url contructor of the source location, combined with the filename.

4.4.2 globus_replica_management_file_delete()

globus_result_t
globus_replica_management_file_delete(
*globus_replica_management_handle_t * handle,*
*char * collection_url,*
*char * location_name,*
*char * filename,*
globus_bool_t delete_file);

Remove filename from the replica catalog location, and if delete_file is GLOBUS_TRUE than also remove the file from the storage system.

4.4.3 globus_replica_management_restart()

TBD: Define function which will get restart/rollback info from rollback lock, and restart the operation…

4.4.4 globus_replica_management_rollback()

TBD: Define function which will get restart/rollback info from rollback lock, and restart the operation…

4.5 Remaining Issues

4.5.1 globus_replica_management_file_update()

The previous version of this spec had a separate file_update() function. The only real difference from file_copy() is that it may employ some optimized algorithm for doing the update, such as modifying only those parts of the file that changed. Perhaps they should be merged into a single function, with arguments controlling the different behavior. Here is the previous text…

The function **globus_replica_management_update_files** updates a set of previously replicated files: that is, it (a) updates a specified set of files on a specified destination file system so that their contents are identical to those at a specified source storage system and (b) updates the timestamp associated with the destination replicas in a specified replica catalog. Notice that this specification permits both simple implementations that simply copy the files in their entirety and more sophisticated implementations that attempt to improve performance by transferring only the modified elements of a file.

This function is not responsible for deciding whether an update is required: it simply updates the specified files. An application that wishes to use this function to update files will typically consult metadata about specific replicas to determine whether "something" has changed: e.g., timestamp or file size or checksum. An application-specific algorithm may be required to obtain this metadata and perform the comparison. Checking to see if a collection of files need to be updated is facilitated by the function

4.5.2 globus_replica_management_file_is_current()

Determine if a file at one location is current with respect to another location.

Arguments to this function are:

- *Replica Catalog.* LDAP URL to the catalog that is managing replica information.
- *File list*. List of logical file names that are to be checked for currency.
- *Source*. Name of the storage system from which data contains the "original" data.
- *Destination*. Name of the storage system to which contains copied data. Name of a storage system as it appears in the replica catalog
- *Comparison Callback.* User-supplied function to be called with name of file to be checked, replica entries for origional and copied file, and name of origional and copy storage system. Function returns –1 if origional is to be updated, 0 if files are equivalent, and 1 if copy need to be updated.
- *Result Vector.* Array of integers that reflect result of calling comparison function on each file.

The function returns a 0 if comparison completed, an error code otherwise.

4.6 Data Transfer Protocol Extensions

We plan to investigate two specific extensions to our GSI-FTP data transfer protocol.

4.6.1 Partial Updates

We propose to extend GSI-FTP to use block checksum information on source and destination copies of a file to determine which blocks need to be transferred during an update operation. Details remain to be worked out, but we are confident that our GSI-FTP implementation will support this.

Notice that this approach provides a particularly elegant integration of partial updates into the replica management process, but that because it involves a protocol extension, it will only be available on extended GSI-FTP servers. In particular, it will most likely not be available in the HPSS pftpd, at least not initially.

Issue: Are checksums stored in the replica catalog? When are they updated? Are these checksums passed as an argument to the GSI-FTP function, or exchanged by that function?

4.6.2 Fault Management and Restart

Various failures can cause an FTP transmission to fail. Error handling capabilities build into GSI-FTP can be used both to specify error-handling strategies (e.g., retries) and to resume interrupted transfers. Only when this error recovery process is unsuccessful will we signal an error via the replica management function error return code.

We note also here another capability that we may wish to incorporate into GSI-FTP in the future. It has been observed that for large data transfers, the 16-bit checksum associated with TCP packets may not be sufficient. To address this problem, we can build data integrity checking into GSI-FTP. (Vern Paxson of LBNL measured that the Internet corrupts 1 out of every 5000 packets[i], and observed that "A corruption rate of 1 packet in 5,000 is certainly not negligible, because TCP protects its data with a 16-bit checksum. Consequently, on average one bad packet out of 65,536 will be erroneously accepted by the receiving TCP, resulting in *undetected data corruption*. If the 1 in 5000 rate is correct, then about one in every 300 million Internet packets is accepted with corruption – certainly many each day." For large data transfers, an IP packet typically contains 64 Kbytes of data. So if one in every 300 million packets is corrupt, then for every 19.2 Terabytes transferred, there will be one undetected error.)

4.6.3 Compression

We could also incorporate support for on-the-fly compression during transfer. However, it is not clear how useful this is.

4.7 Implementation Approach

We propose to prototype our replica management functions as Perl scripts, in order to permit rapid exploration of alternative strategies. Second choice of implementation is to provide a Java binding.

4.8 Storage System Requirements

We require the following capabilities in any storage system to be used by our replica management system:

- GSI-FTP support, hence enabling single sign-on access for users with suitable GSI credentials. This requirement implies a need for a GSI-FTP-enabled server on the storage system (GSI-FTP-enabled servers exist for regular FTP, HPSS, and Unitree to date). The GSI-FTP server must be configured so as to accept the certificate authorities used by the particular user community being supported.
- Information service support (specifically, a storage system Grid Resource Information Service, or GRIS), so that remote users can enquire about storage system performance, available storage space, etc.
- Logging of all transactions for audit purposes.
- User and group level access control on a per-file level. Access control should be with respect to distinguished names contained in the certificate used to authenticate to the storage system. Group membership will be provided via a group authentication service. Initial implementation of this will be simply a group file that will be maintained by each experiment and distributed to each storage system using GSI-FTP.

In the future, we will also want the storage system to support advance reservation of space and bandwidth.

4.9 Consistency of Replicated Files

Our proposed replica management system layers on top of other data management systems. Hence, we cannot guarantee the persistence or integrity of any replicas that we create. As a result of, for example, failure, the actions of another user with appropriate privileges, or an automated cache management system, replicas may be deleted or modified.

Such problems can be mitigated but not entirely prevented via techniques such as the following:

- *Access control*. By denying delete and update access to other users, one can ensure that the replicated files will not be removed or modified on the target system. Note however, that in the situation that the target is managed as a data cache, system policy may delete files even if delete access is denied to other users.
- *Advisory locks*. Additional attributes could be associated with replicas indicating that they are currently "in use." It would be up to community convention not to delete files with this advisory lock set.
- *Replica time to live*. A refinement of the above strategy, this method associates a time-to-live value with each replica. This value could be updated periodically. The time to live approach is more resilient to application failure than advisory locks.

In the case of Objectivity applications, we also observe that in the situation in which remote database references are used, a replication operation may change the contents of a catalog for a file that is in use. Such changes have been demonstrated to crash Objectivity applications. Thus the local import functions must lock the catalog, perform the catalog update as an atomic transaction, and then notify all active applications to re-read the catalog so as to have an accurate view of the federation state.

5 Ideas for Phase 2 Development

We suggest here some of the features that might be incorporated into higher-level functions during Phase 2 of this project. Note that in contrast to Phase 1 activities, some of the functionality required here may well be application-dependent.

Incorporation of advance reservation. The Globus architecture addresses advance reservation issues via its General-Purpose Architecture for Reservation and Allocation (GARA) system. With appropriate support within storage systems, we can, for example, ensure that there is sufficient space at a destination storage system for a transfer to complete successfully.

Automatic selection of replica source locations. We can imagine a variant of the replica creation function that does not require a "source" as an argument: instead, it consults to replica catalog to determine where replicas are located, consults GIS to determine relevant properties of those locations (e.g., transfer speeds), and then performs copies from the "best" locations. Semi-automatic variants can also be defined. If these techniques are used widely, then the performance and scalability of the replica catalog becomes a significant concern.

Automatic selection of replica destination locations. Similarly, we can imagine a function that monitors data access patterns and generates new replicas at selected locations in order to reduce overall network load.

Support movement of complete logical collections. We have been asked to extend our replica management API to include the function "Copy a complete logical collection to a specified location." Such a function might require accessing multiple source locations to find all the files in the logical collection.

6 Proposed Experimental Approach

In order to gain early experience with the replica management functions described above, we propose to experiment with their use within the BaBar experiment. We anticipate that this experience in an operational setting will enable us to converge rapidly on a solid design that can then be deployed more widely.

A first experiment will be simply to demonstrate the ability to move a single database file from one Objectivity server to another using the replica management API. This step will require implementing a subset of the functionality specified for the register, create, and delete functions. We will deploy an initial replica catalog at ISI and two GSI-FTP servers, one at SLAC and the other in France. Database catalog import and export functions will also need to be provided.

7 Complementary Activities

- We note that the activities planned here enable a wide variety of complementary activities. We hope to work with other groups to realize capabilities such as those listed in Section 5 above as well as the following.
- *Improved logging*. The basic logging capabilities that we require in individual storage systems can be extended to provide for distributed analysis of data access patterns. Integration with logs generated by the replica management service itself and with performance measurement tools will allow for monitoring and improvement of performance. It will be desirable for logs to be maintained on stable and secure storage in order to enable their use by intrusion detection systems.
- *Improved fault recovery*. The replica management service should be improved over time to incorporate increasingly sophisticated fault recovery. For example, it may be desirable to be able to recover from a total loss of the contents of the replica catalog, via reconstruction of the catalog from log records.
- *Improved performance measurement*. Our Phase 1 deliverables will incorporate some basic performance measurement capabilities, but achieving consistently high performance across transcontinental and intercontinental networks will require more sophisticated measurement and monitoring support.
- *Integration of dynamic container creation*. As noted above, some groups are interested in replication at the object level, via the creation and distribution of "dynamic containers."
- *End-to-end scheduling*. Reservation capabilities in the network as well as in storage systems will allow for end-to-end scheduling of replica creation operations.

Acknowledgments

This document reflects the results of a number of fruitful discussions:

- In Pittsburgh, involving Bill Allcock, Ann Chervenak, Ian Foster, Andy Hanushevsky, Koen Holtman, Carl Kesselman, Asad Samar, Steven Tuecke, and others.
- At Argonne National Laboratory, involving Ann Chervenak, Ian Foster, Wolfgang Hoschek, Carl Kesselman.
- At SLAC, involving Jacob Becla, Ann Chervenak, Ian Foster, Andy Hanushevsky, Carl Kesselman, Bob Jacobsen, Richard Mount, Harvey Newman, Charles Young.
- Via email with numerous people including those named above and Fabrizio Gagliardi.

[i] V. Paxson, End-to-End Internet Packet Dynamics, SIGCOMM '97, LBNL-40488 (ftp://ftp.ee.lbl.gov/papers/vp-pkt-dyn-sigcomm97.ps)